SOL—A Symbolic Language for General-Purpose Systems Simulation

D. E. KNUTH AND J. L. McNELEY

Summary—**This paper illustrates the use of SOL, a general-purpose algorithmic language useful for describing and simulating complex systems. Such a system is described as a number of individual processes which simultaneously enact a program very much like a computer program. (Some features of the SOL language are directly applicable to programming languages for parallel computers, as well as for simulation.) Once a system has been described in the language, the program can be translated by the SOL compiler into an interpretive code, and the execution of this code produces statistical information about the model. A detailed example of a SOL model for a multiple on-line console system is exhibited, indicating the notational simplicity and intuitive nature of the language.**

SIMULATION by computer is one of the most important tools available to scientists and engineers who are studying complex systems. The first computer programs of this type were especially designed to simulate some particular model; but afterwards the authors of several of these programs abstracted the essential features of their program organization and prepared *general-purpose* simulation programs. The most extensively used general-purpose programs of this type have apparently been the SIMSCRIPT compiler of Markowitz, Hauser, and Karr [1], and the GPSS (General-Purpose Systems Simulator) routines of Gordon [2]–[4].

Although SIMSCRIPT and GPSS are both general-purpose simulation programs, they are built around quite different concepts because of their independent evolution, and so they bear little resemblance to each other. SOL (Simulation-Oriented Language) is another general-purpose simulation routine, in which we have attempted to incorporate the best features of the other languages. After a careful study of SIMSCRIPT and GPSS, and after having implemented a version of GPSS for another computer, we found that it would be possible to generalize the characteristics of the former programs, while at the same time the language became simpler and more convenient for the preparation of models. This simplification was achieved by extracting the essential characteristics of GPSS and recasting them into a symbolic language such as SIMSCRIPT. There are, of course, a great many ways in which this can be done, and we are not sure that the compromises we have chosen have been optimal; but a year of experience with the SOL language, after applying it to a number of problems of different kinds, indicates that SOL is a quite powerful and flexible way to describe systems for simulation. We also found that the increased generality available in SOL was actually simpler to implement into a computer program than the previous routines were.

A complex system can be represented as a number of individual processes, each of which follows a *program* very much like a computer program. For example, if we were simulating traffic in a network of streets, we might have one program describing a typical automobile (or perhaps two programs, one which describes all of the women drivers and one which describes all of the men), another program which represents the action of traffic signals, and possibly some other programs representing pedestrians, etc. Each program depends not only on quantities which are specified in advance, but also on *random* quantities which describe a probabilistic behavior; thus, we can specify the probability that a driver will turn left, the probability that he will switch lanes, the distribution of speeds, etc. Although each program represents only a single entity (such as a single automobile), there can be many entities each carrying out the same program, each at its own place in the program.

Because of these considerations, SOL is a language which is in many respects very much like a problem-oriented language such as ALGOL or FORTRAN. There are three major points of difference between SOL and conventional compiler languages. SOL provides

1) mechanisms for parallel computation,
2) a convenient notation for random elements within arithmetic expressions,
3) automatic means of gathering statistics about the elements involved.

On the other hand, many of the features of problem-oriented languages do not appear in SOL, not because they are incompatible with it, but rather because they introduce more complication into this scheme than seems to be of practical value for simulation processes.

A program written in the SOL language is punched onto cards and it is then compiled by the SOL *compiler* into an interpretive pseudocode. The SOL *interpreter* is another machine program, which executes this pseudocode and produces the results. (The SOL system has been implemented for the B5000 computer, but at the present time it is being used only for research within the Burroughs Corporation, and it is not currently available for distribution.)

A self-contained, complete description of SOL ap-

Manuscript received January 3, 1964.
D. E. Knuth is with the California Institute of Technology, Pasadena, Calif.
J. L. McNeley is with the Burroughs Corporation, Pasadena, Calif.

pears in another paper [5]. The definition there is rather terse since it is intended primarily as a reference description; we will introduce the language here by means of an example, discussing the significance of each statement in an intuitive fashion.

Example: Communication with Remote Terminals

The following example has been chosen not only to illustrate most of the features of SOL, but also because it is a practical application in which SOL has been used to evaluate the design of an actual system of some complexity.

Consider the configuration shown in Fig. 1. This represents one of four similar groups of devices which all share the processor shown at the right. The "TU's" are terminal units which may be thought of as inquiry stations or typewriters. There are three groups of typewriters, with three in the first group (TU[1], TU[3], TU[5]), two in the second group (TU[2], TU[4]) and only one in the third (TU[6]). These groups are located many miles from each other and from the central processor. People come in at the rate of about five or six per minute to use each typewriter, and they wait in the appropriate queue until the typewriter is free.

These people will send one of three kinds of messages.

Message	Frequency	Compute time	Number of Response Words
A	20 per cent	250 msec	3
B	50 per cent	300 msec	4
C	30 per cent	400 msec	5

Each message type has a different frequency and requires a different amount of central processor time.

Communication between the typewriters and the processor is handled by *site buffers* SB[1], SB[2], SB[3], one at each remote site, and by two *processor buffers* PBU's, which receive the information and transmit it to the computer. These processor buffers sequentially scan TU[1], TU[2], · · · , TU[6], TU[1], · · · until locating a typewriter ready to transmit information; this scanning is done by sending control pulses to all lines, then receiving a "positive" response from the SB if the appropriate TU is ready. Then a message is transferred from SB to the PBU and from there to the processor; after computing the answer, the processor refills the PBU, and the appropriate number of words is sent back to the SB and is typed on the TU (one word at a time). Further details will be given as we discuss the program.

We will compose three programs.

1) A program which describes the action of each person who uses the remote typewriters.
2) A program which describes the action of each of the two PBU's.
3) A program which simulates the action of the other six PBU's, which share the central processor with the configuration shown in Fig. 1.

Fig. 2 shows these three programs together with the control information, as a complete SOL model.

The independent quantities which enact the programs as the simulation proceeds are called *transactions*. (Much of the terminology used in SOL is taken from Gordon's simulator [2]–[4].) As simulation begins, there are only three transactions: one for each of the programs 1), 2), 3). Therefore, these programs describe not only the action of the quantities mentioned above, they also describe the creation and dissolution of new transactions.

Each transaction contains *local variables* which have values that can be referred to only by that transaction. There are also *global variables*, and some other types of global quantities, which can be referred to by all transactions. Thus, transactions can interact with each other by setting and testing global quantities. Only one "copy" of each global variable is present in the system, but there are in general many copies of each local variable (one for each transaction).

Program 1), which represents the people using the typewriters, might begin as follows:

```
process USERS;
begin integer Q, START TIME, MESSAGE TYPE;
new transaction to START; new transaction to START;
ORIGIN: new transaction to START; wait 0:5000; go to
    ORIGIN;
START:
```

The first line merely identifies a *process* (*i.e.*, a program) with the name "USERS." The language resembles ALGOL, and we distinguish control words by putting them in bold-face type. The second line states that there are three local variables in these transactions, having the names Q, START TIME and MESSAGE TYPE. The statement "**new transaction to** START" describes the creation of a new transaction whose local variables have the same values as the local variables of the parent transaction (in this case zero, since all local variables are automatically set to zero at the beginning of a process), and this new transaction begins executing the program at the statement labeled START. The statement "**wait** 0:5000" means an amount of simulated time, chosen randomly from 0 to 5000, is to elapse before the next statement is executed. In general, the statement "**wait** E," where E is some expression, means that E units of time are to pass before excuting the next statement. The expression $E_1:E_2$ always denotes a random integer chosen between E_1 and E_2, and therefore "**wait** 0:5000" has the meaning stated above. A unit of time in this case represents 1 msec in the simulated model.

The reader should now reread the above sequence of coding before proceeding further. The essential action it describes is that three transactions will begin executing the program beginning at the statement called START, and thereafter a new transaction (*i.e.*, a new user enter-

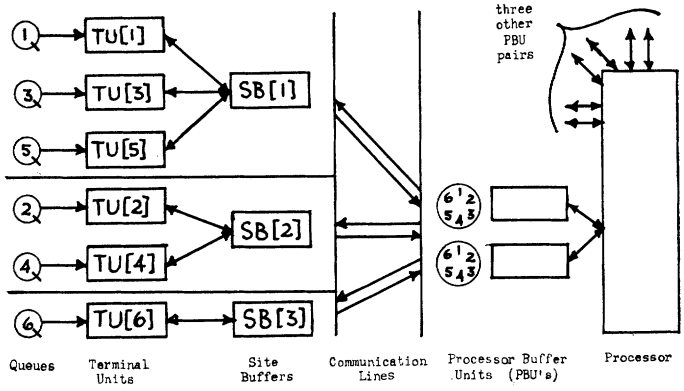

Fig. 1—Multiple console on-line communication system.

```
begin
facility TU[6], SB[3], LINE, COMPUTER;
store 10 QUEUE[6];
integer TUSTATE[6], SBNUMBER[6], TUMESSAGE[6];
table (2000 step 500 until 15000) TABLE[6];
process MASTER CONTROL;
begin SBNUMBER[1]←1; SBNUMBER[2]←2;
      SBNUMBER[3]←1; SBNUMBER[4]←2;
      SBNUMBER[5]←1; SBNUMBER[6]←3;
wait 60×60×1000; stop end;
process USERS;
begin integer Q, START TIME, MESSAGE TYPE;
new transaction to START; new transaction to START;
ORIGIN: new transaction to START; wait 0:5000; go to
  ORIGIN;
START: Q←1:6; enter QUEUE[Q];
MESSAGE TYPE←(1,1,2,2,2,2,2,3,3,3);
seize TU[Q];
TUMESSAGE[Q]←MESSAGE TYPE;
wait 6000:8000;
START TIME←time;
output #TU#, Q, #SENDS MESSAGE#, MESSAGE TYPE,
  #AT TIME#, time;
TUSTATE[Q]←1;
wait until TUSTATE[Q]=0;
release TU[Q]; leave QUEUE[Q];
tabulate (time−START TIME) in TABLE[Q];
output #TU#, Q, #RECEIVES REPLY AT TIME#, time;
cancel end;
process PBU; begin integer S, T, WORDS;
new transaction to SCAN; T←3;
SCAN: T←T+1; if T>6 then T←1; wait 1;
S←SBNUMBER[T];
seize LINE;
wait 5; if SB[S] busy then (wait 80; release LINE; go to
  SCAN);
seize SB[S]; wait 15; if TUSTATE[T]≠1 then
(wait 65; release LINE; release SB[S]; go to SCAN);
wait 225; SEND: wait 170; if pr(0.02) then (wait 20; go to
  SEND);
new transaction to COMPUTATION; wait 20; release SB[S];
release LINE; TUSTATE[T]←2; cancel;
COMPUTATION: seize COMPUTER; WORDS←TUMESSAGE[T]
  +2;
wait (if WORDS=3 then 250 else if WORDS=4 then 300
  else 400);
release COMPUTER;
OUTPUT: wait 1; seize LINE; wait 5;
if SB[S] busy then (wait 80; release LINE; go to OUTPUT);
seize SB[S]; wait 75;
RECEIVE: wait 80; if pr(0.01) then (wait 20; go to
  RECEIVE);
release LINE;
WORDS←WORDS−1;
if WORDS=0 then new transaction to SCAN;
wait 325; release SB[S]; wait 170;
if WORDS>0 then go to OUTPUT;
TUSTATE[T]←0; cancel end;
process OTHER PBUS;
begin integer I; I←6;
CREATE: new transaction to COMPUTE;
I←I−1; if I>0 then go to CREATE; cancel;
COMPUTE: wait 3200:5000; seize COMPUTER;
wait (250, 250, 300, 300, 300, 300, 300, 400, 400, 400);
release COMPUTER; go to COMPUTE end;
end.
```

Fig. 2—Complete SOL program for the on-line system.

ing the system) will be created at intervals of about 2.5 sec. We have started the system with three transactions so that it will not take it very long to arrive at a more or less stable condition.

The program now proceeds as follows:

START: Q←1:6; **enter** QUEUE[Q];

The statement "Q←1:6" means that local variable Q is set to a random number between 1 and 6; thus the user is assigned to one of the six typewriters. The "enter" statement refers to one of six global quantities, QUEUE[1], · · · , QUEUE[6]. At the conclusion of the simulation, data will be reported giving the average number of people in each queue at a given time, and also the maximum number.

MESSAGE TYPE←(1,1,2,2,2,2,2,3,3,3);

The expression $(E_1, E_2, \cdots, E_n)$ denotes a random choice selected from among the n expressions. Therefore, the given statement means that the local variable MESSAGE TYPE receives the value 1 with probability 20 per cent, 2 with probability 50 per cent and 3 with probability 30 per cent; this represents the choice of message A, B or C as stated earlier.

seize TU[Q];

This statement refers to one of the global quantities TU[1], · · · , TU[6], which are classified as *facilities*. A facility is *seized* by one transaction, and then it cannot be seized by another transaction until it has been *released* by the former transaction. Therefore, if transaction X comes to a seize statement, where the corresponding facility is *busy* (*i.e.*, has been seized by transaction Y), transaction X stops executing its program until transaction Y releases the facility. If several transactions are waiting for this event, they are processed in a first-come-first-served fashion.

Thus, the statement "**seize** TU[Q]" expresses the situation that the user takes control of typewriter number Q, after possibly waiting in line for it to become available.

TUMESSAGE[Q]←MESSAGE TYPE;

This statement says that the global variable TUMESSAGE[Q] is set to indicate the type of message. This global variable is used to communicate with the PBU process which is described below.

wait 6000:8000;

This statement simulates the time of 6 to 8 sec, taken by the man to type his request on the terminal unit.

START TIME←**time;**

We now set the local variable START TIME equal to "**time,**" the current value of the simulated clock.

output #TU#, Q, #SENDS MESSAGE#, MESSAGE TYPE, #AT TIME#, **time;**

This statement causes the printing of a line during the simulation, having the form "TU 3 SENDS MESSAGE 2 AT TIME 12610." The "#" symbols indicate a string inserted into the output.

TUSTATE[Q]←1;

Another global variable TUSTATE[Q] is now set to 1 to indicate that the typed message is ready to send. TUSTATE[Q] has three possible settings.

TUSTATE = 0 means the TU is free.
TUSTATE = 1 means the message has been typed.
TUSTATE = 2 means the answer message may be typed.

The next statement

wait until TUSTATE[Q] = 0;

means the transaction is to stop at this point until TUSTATE[Q] has been set to zero (by some other transaction). This indicates that we are to wait until the answer message has been fully received. When that occurs, the transaction finishes its work as follows:

release TU[Q]; **leave** QUEUE[Q];
tabulate (**time** − START TIME) **in** TABLE[Q];

The latter statement is used for statistical data; TABLE[Q] is a global quantity which receives "readings" by means of "tabulate" statements. At the end of simulation, this table is printed out giving the mean, the standard deviation and a histogram of the data it has received.

output #TU#, Q, #RECEIVES REPLY AT TIME#, **time;**
cancel end;

The last statement, "**cancel,**" causes the disappearance of the transaction, and the word "**end**" indicates the end of the program for this process.

Program 2), which runs simultaneously with 1) and 3), describes the action of the PBU's.

process PBU; **begin integer** S, T, WORDS;
new transaction to SCAN; T←3;
SCAN:

We have three local variables, S, T and WORDS. At the beginning, two transactions (representing the two PBU's) start at SCAN, one with its variable T = 0, the other with T = 3.

SCAN: T←T+1; **if** T > 6 **then** T←1; **wait** 1;

These statements represent the cyclic scanning process which we assume takes 1 msec. The variable T represents the number of the TU which the PBU will be referencing.

S←SBNUMBER[T];

"SBNUMBER" is a table of constants, which is used to tell which SB corresponds to the TU scanned.

seize LINE;

We now seize the facility LINE, which represents the long-distance communication lines. (If the other PBU has seized LINE already, we must wait until it has been released.)

```
wait 5; if SB[S] busy then
                    (wait 80; release LINE; go to SCAN);
```

We wait 5 msec for a control signal to propagate to the SB unit. Here SB[S] is a facility; if it is busy (*i.e.*, has been seized by the other PBU) we wait 80 msec more, receiving no signal back, so we release the line and return to scan the next TU.

```
seize SB[S]; wait 15; if TUSTATE[T]≠1 then
(wait 65; release LINE; release SB[S]; go to SCAN);
```

If SB[S] received the control signal, it is brought under the control of this PBU. Fifteen milliseconds later, the number T has been transmitted across the line, and it takes 65 msec for the SB to determine if TU[T] is ready to transmit or not. If not, we release the SB and the line, and scan again.

```
wait 225; SEND: wait 170; if pr(0.02) then
                    (wait 20; go to SEND);
```

It takes 225 msec for the SB to get ready to transmit the message and to send a warning signal across the line to the PBU. Then 170 msec are required to send the input message. The construction "**if** pr(0.02)" means "2 per cent of the time," and so this statement indicates that, with probability 0.02, a parity error in the transmission is detected; in such a case, we send back a signal calling for retransmission of the message.

```
new transaction to COMPUTATION; wait 20; release SB[S];
release LINE; TUSTATE[T]←2; cancel;
```

At this point two parallel processes take place. As the PBU tries to send the message to the computer, it also sends a "message received" signal across the lines to the SB, and, 20 msec later, the SB and the lines are released. The TUSTATE is adjusted, and then this portion of the transaction is cancelled.

```
COMPUTATION: seize COMPUTER;
  WORDS←TUMESSAGE[T]+2;
wait (if WORDS=3 then 250 else
  if WORDS
  =4 then 300 else 400);
release COMPUTER;
```

Here we send the message to the computer facility, possibly waiting for it to become available. The local variable WORDS is set to the number of words output for the current message, and we also wait the appropriate amount of computer time. At this point, the output message has been created by the computer, and it has been sent back to the PBU. The final job is to output this message, one word at a time:

```
OUTPUT: wait 1; seize LINE; wait 5;
if SB[S] busy then (wait 80; release LINE; go to OUTPUT);
```

A control word is sent out to interrogate the SB, as in the case of input above.

```
seize SB[S]; wait 75;
RECEIVE: wait 80; if pr(0.01) then
  (wait 20; go to RECEIVE);
release LINE;
```

We have output one word to the SB; there was probability 1 per cent that a transmission error was detected.

```
WORDS←WORDS−1;
if WORDS=0 then new transaction to SCAN;
wait 325; release SB[S]; wait 170;
```

After the last word has been transmitted, a parallel activity starts with another scan. It takes 325 msec for the SB to send the word to the typewriter, and another 170 msec are required for the typewriter to finish its typing.

```
if WORDS>0 then go to OUTPUT;
TUSTATE[T]←0; cancel end;
```

When the output has all been typed, TUSTATE is reset to zero (thus activating the USER transaction) and this parallel branch of the program disappears.

Program 3) is used to describe the traffic which takes place at the computer, by creating six simulated PBU's as follows:

```
process OTHER PBUS;
begin integer I; I←6;
CREATE: new transaction to COMPUTE;
I←I−1; if I>0 then go to CREATE; cancel;
COMPUTE: wait 3200:5000; seize COMPUTER;
wait (250,250,300,300,300,300,300,400,400,400);
release COMPUTER; go to COMPUTE end;
```

Our example program is now almost complete. We precede the three processes given above by the following code, which declares the global quantities. There is also a fourth process which accomplishes the initialization and which stops the simulation after 1 hour of simulated time.

```
facility TU[6], SB[3], LINE, COMPUTER;
store 10 QUEUE[6];
integer TUSTATE[6], SBNUMBER[6], TUMESSAGE[6];
table (2000 step 500 until 15000) TABLE [6];
process MASTER CONTROL;
begin SBNUMBER[1]←1; SBNUMBER[2]←2;
      SBNUMBER[3]←1; SBNUMBER[4]←2;
      SBNUMBER[5]←1; SBNUMBER[6]←3;
wait 60×60×1000; stop end;
```

Remarks

We have purposely chosen a rather complex example to show how SOL can be used to solve an actual problem of practical importance, and to show in what a natural manner the system can be described in the language.

Fig. 3 is a sample of some of the output resulting from the program of the preceding section.

```
TU          6    SENDS MESSAGE          1     AT TIME        6586
TU          4    SENDS MESSAGE          1     AT TIME        7152
TU          5    SENDS MESSAGE          2     AT TIME        7295
        TU          6     RECEIVES REPLY AT TIME        9973
        TU          4     RECEIVES REPLY AT TIME       10305
        TU          5     RECEIVES REPLY AT TIME       13353
TU          6    SENDS MESSAGE          3     AT TIME       16908
TU          2    SENDS MESSAGE          2     AT TIME       17476
TU          5    SENDS MESSAGE          1     AT TIME       19405
        TU          6     RECEIVES REPLY AT TIME       21166
        TU          2     RECEIVES REPLY AT TIME       21412
TU          3    SENDS MESSAGE          2     AT TIME       21646
        TU          5     RECEIVES REPLY AT TIME       24229
TU          1    SENDS MESSAGE          2     AT TIME       25424
        TU          3     RECEIVES REPLY AT TIME       27959
TU          4    SENDS MESSAGE          1     AT TIME       30442
TU          5    SENDS MESSAGE          2     AT TIME       31409
        TU          1     RECEIVES REPLY AT TIME       31609
        TU          4     RECEIVES REPLY AT TIME       33278
TU          3    SENDS MESSAGE          3     AT TIME       34067
TU          2    SENDS MESSAGE          3     AT TIME       34478
        TU          5     RECEIVES REPLY AT TIME       35046
        TU          2     RECEIVES REPLY AT TIME       38958
        TU          3     RECEIVES REPLY AT TIME       39376
TU          1    SENDS MESSAGE          1     AT TIME       40472

CLOCK TIME AT END OF SIMULATION WAS   3600000

NUMBER OF TIMES LABELS WERE ENCOUNTERED

LABEL          -      COUNT      LABEL          -      COUNT      LABEL          -      COUNT
      ORIGIN   -       1455            START    -       1457             SCAN   -      17303
        SEND   -       1477      COMPUTATION    -       1446           OUTPUT   -       8021
     RECEIVE   -       5990           CREATE    -          6          COMPUTE   -       4764

                    NAME OF FACILITY     FRACTION OF TIME IN USE

                          TU[001]                0.8318
                          TU[002]                0.8055
                          TU[003]                0.7887
                          TU[004]                0.8085
                          TU[005]                0.8302
                          TU[006]                0.7585
                          SB[001]                0.6051
                          SB[002]                0.4221
                          SB[003]                0.2120
                             LINE                0.8649
                         COMPUTER                0.5509
```

NAME OF STORE	CAPACITY	MAXIMUM USED	AVERAGE OCCUPANCY	AVERAGE UTILIZATION
QUEUE[001]	10	10	2.5272	0.2527
QUEUE[002]	10	10	2.4255	0.2426
QUEUE[003]	10	10	2.3835	0.2384
QUEUE[004]	10	7	1.7696	0.1770
QUEUE[005]	10	8	2.1844	0.2184
QUEUE[006]	10	5	1.4971	0.1497

TABLE NAME IS TABLE[003]

NUMBER OF TABLE ENTRIES 235 SUM OF ALL ENTRY VALUES 1201076

MEAN OF TABLE 5110.9617 STANDARD DEVIATION 1441.51124

UPPER LIMIT	NUMBER	PER CENT	CUMULATIVE	MULTIPLE OF MEAN
2000	0	0.00	0.00	0.3913
2500	0	0.00	0.00	0.4891
3000	5	2.13	2.13	0.5870
3500	14	5.96	8.09	0.6848
4000	36	15.32	23.40	0.7826
4500	32	13.62	37.02	0.8805
5000	46	19.57	56.60	0.9783
5500	23	9.79	66.38	1.0761
6000	25	10.64	77.02	1.1739
6500	18	7.66	84.68	1.2718
7000	12	5.11	89.79	1.3696
7500	10	4.26	94.04	1.4674
8000	5	2.13	96.17	1.5653
8500	1	0.43	96.60	1.6631
9000	4	1.70	98.30	1.7609
9500	1	0.43	98.72	1.8587
10000	1	0.43	99.15	1.9566
10500	1	0.43	99.57	2.0544
11000	0	0.00	99.57	2.1522
11500	1	0.43	100.00	2.2501
12000	0	0.00	100.00	2.3479
12500	0	0.00	100.00	2.4457
13000	0	0.00	100.00	2.5436
13500	0	0.00	100.00	2.6414
14000	0	0.00	100.00	2.7392
14500	0	0.00	100.00	2.8370
15000	0	0.00	100.00	2.9349

Fig. 3 (*pp. 406–407*)—Samples of the output obtained.

The ideas used in SOL for creating and canceling transactions have applications in the design of languages for highly parallel computers.

The techniques which are used in the implementation of SOL will be the subject of another paper. It should be indicated here, however, that the implementation gives a rather efficient program because separate lists are kept for transactions which are waiting for different reasons. Those which are waiting for time to pass are kept sorted on the required time. Those which are waiting for a condition such as "wait until A = 0," for some global variable A, are kept in a list associated with A; this list is interrogated only when the value of A has been changed.

The SOL system has proved to be especially advantageous for simulating computer systems since "typical programs," which we assume are to be run on the simulated computers, are easily coded in SOL's language.

ACKNOWLEDGMENT

The authors wish to express their appreciation to J. Merner for many helpful suggestions.

REFERENCES

[1] H. M. Markowitz, B. Hauser, and H. W. Karr, "SIMSCRIPT—A Simulation Programming Language," Prentice-Hall, Inc., Englewood Cliffs, N. J.; 1963.
[2] G. Gordon, "A general purpose systems simulation program," *Proc. Eastern Joint Computers Conf.*, pp. 87–104; December, 1961.
[3] ——, "A general purpose systems simulation program," *IBM Systems J.*, vol. 1, pp. 18–32; September, 1962.
[4] "Reference Manual, General Purpose Systems Simulator II," IBM Corp., White Plains, N. Y.; 1963.
[5] D. E. Knuth and J. L. McNeley, "A formal definition of SOL," this issue, page 409.
[6] M. R. Lackner, "Toward a general simulation capability," *Proc. Spring Joint Computer Conference*, pp. 1–14; May, 1962.